Caching Dynamic Web Content: Designing and Analysing an Aspect-Oriented Solution

Sara Bouchenak[1], Alan Cox[2], Steven Dropsho[3],
Sumit Mittal[4⋆], Willy Zwaenepoel[3]
[1] *INRIA, 655, av. de l'Europe, Montbonnot, 38334 St. Ismier Cedex, France*
Sara.Bouchenak@inria.fr
[2] *Rice University, Department of Computer Science, Houston, TX USA*
alc@cs.rice.edu
[3] *EPFL, Department of Computer Science, CH-1015 Lausanne, Switzerland*
{Steven.Dropsho, Willy.Zwaenepoel}@epfl.ch
[4] *IBM India Research Lab, Block-1, IIT, Hauz Khas, New Delhi, India*
sumittal@in.ibm.com

Abstract. Caching dynamic web content is an effective approach to reduce Internet latency and server load. An ideal caching solution is one that can be added transparently by the developers and provides complete consistency of the cached documents, while minimizing false cache invalidations. In this paper, we design and implement AutoWebCache, a middleware system for adding caching of dynamic content transparently to J2EE server-side applications having a backend database. For this purpose, we first present the principles involved in caching dynamic web content, including our logic to ensure consistency of the cached entries. Thereafter, we demonstrate the use of aspect-oriented (AOP) techniques to implement our system, showing how AOP provides modularity and transparency to the entire process. Further, we evaluate the effectiveness of AutoWebCache in reducing response times of applications, thereby improving throughput. We also analyze the transparency of our system for a general application suite, considering issues such as dynamic web pages aggregating data from multiple sources, presence of insufficiently structured interfaces for exchanging information and the use of application semantics while caching. We use two standard J2EE web benchmark applications, RUBiS and TPC-W, to conduct our experiments and discuss the results obtained.

Keywords: Caching, aspect-oriented programming, J2EE applications, dynamic content.

1 Introduction

Dynamically generated web content represents a large portion of web requests. The rate at which dynamic documents are delivered is often orders of magnitudes slower than static documents [9, 11]. Therefore, caching dynamic web content is

⋆ Work done while being at Rice University, Houston and EPFL University, Lausanne

an appealing approach to reduce Internet latency and server load. Web sites for dynamic content are usually based on a multi-tier J2EE architecture using several middleware systems [27]: an HTTP server as a web front-end and provider of static content, an application server to execute the business logic and generate the dynamic web content, and a database to store the persistent data required by the application. Dynamic content generation places a significant burden on the servers, often leading to performance bottlenecks. Caching dynamic web content can directly address these bottlenecks.

Implementing caching as a middleware solution is particularly attractive. Of course, an ideal solution is one that can be added transparently by the developers, possibly even as an after-thought. Some examples of transparently adding caching to an application are given in [17, 6, 4], but these ignore consistency of the cached entries. Other solutions provide consistency, but ignore transparency, requiring manual insertion [10]. There are some projects that provide both consistency and transparency, such as those caching SQL query result sets [8] at the back-end. The interesting property of data from result sets of SQL queries is that it is from a single interface and hence, of one type (homogeneous). An open question is whether similar techniques can be successful for more complex content such as web pages that aggregate data from multiple sources (*i.e.*, heterogeneous).

In this paper, we present the design and implementation of *AutoWebCache*, a middleware solution for caching dynamically generated content in J2EE applications. A goal is to move the caching as far forward in the multi-tier architecture to not only reduce the database activity in the back-end but also the business logic activity, which is becoming ever more complex and costly at the middle tier. Unlike caching data such as JDBC SQL results at a single well-specified interface, caching fully formed web pages requires interfacing to both the front-end (e.g., Tomcat servlet engine) *and* the back-end (e.g., JDBC interface). Caching at this level requires information from both interfaces to maintain consistency of the cached documents. To keep the caching transparent, we cast caching as an aspect of the application and use an aspect oriented programming (AOP) framework to capture the information flowing through various interfaces. We give details of the AutoWebCache cache system based on AOP principles and the AspectJ [2] weaving rules that add the caching logic transparently to the application.

We evaluate the performance of our middleware solution with the help of two J2EE benchmarks - RUBiS and TPC-W. RUBiS implements the core functionality of an auction-site: selling, browsing and bidding [1], while TPC-W simulates an online-bookstore [30]. We demonstrate the gains in response times using AutoWebCache for each. We also analyze the transparency of AutoWebCache for a general application suite. We argue that for the general case, issues can arise when caching dynamic content at the front-end due to 1) dynamic web pages aggregating data from multiple sources, 2) some sources not having sufficiently structured interfaces for exchanging information and 3) the need to consider semantics of the application while caching. Although our benchmark applications are servlets-based and use SQL queries to incorporate dynamism, we believe that the results and arguments presented in this paper hold true for a general architecture as well.

The contributions of this paper can be summarized as follows:

1. Design, implementation and evaluation of AutoWebCache, a middleware solution that caches dynamic web pages at the front-end while maintaining consistency with the back-end database(s).
2. Demonstrating that dynamic web caching can be considered a crosscutting aspect and, therefore, AOP methods should be considered as a flexible and easy-to-use tool to develop the middleware support.

The remainder of this paper is organized as follows. Section 2 gives some background on dynamic web applications and aspect-oriented programming. Section 3 outlines the principles involved in designing a dynamic web cache and gives an overview of our AutoWebCache system. Section 4 describes the implementation of AutoWebCache using aspect-oriented techniques, and analyzes its transparency with respect to an application. Sections 5 and 6 present our evaluation environment and the results of our evaluation, respectively. Section 7 provides a discussion of our experiences. Section 8 discusses some related work and finally, Section 9 draws our conclusions.

2 Background

2.1 J2EE Web Applications

Java 2 Platform, Enterprise Edition (J2EE) defines a model for developing distributed applications, *e.g.*, web applications, in a multi-tiered architecture [27]. Such applications usually start with requests from web clients that flow through an HTTP server front-end and provider of static content, then to an application server to execute the business logic and generate web pages on-the-fly, and finally to a database that stores resources and data (see Figure 1).

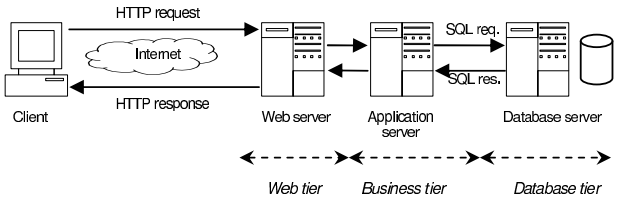


Fig. 1. Architecture of Dynamic Web Applications

Upon an HTTP client request, either the request targets a static web document that the web server can return directly; or the request refers to a dynamic document, in which case the web server forwards that request to the application server. The application server runs one or more software components (e.g., Servlets, EJB) that query a database through a JDBC driver (Java DataBase Connection driver) [28] and retrieve data to generate a web document on-the-fly.

2.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a methodology with concepts and constructs to modularize crosscutting concerns (i.e., *aspects*) [15]. With AOP, the different aspects involved in a system are separately implemented in different modules. The developer can also specify the manner in which these modules need to be woven to form the final system. Aspects are woven together via the *join point* model, a fundamental concept in AOP specifying identifiable execution points in a system. Such join points include method calls and executions, constructor calls, read and write access to fields, exception handler invocations, etc. *Pointcuts* allow a programmer to capture certain join points while an *advice* provides a way to express crosscutting actions to be performed at a certain pointcut. At a pointcut, an advice specifies the weaving rules involving that point, such as performing some actions *before* or *after* the execution of the pointcut. Figure 2 shows the basic principle of adding caching transparently to a web application, using aspect weaving.

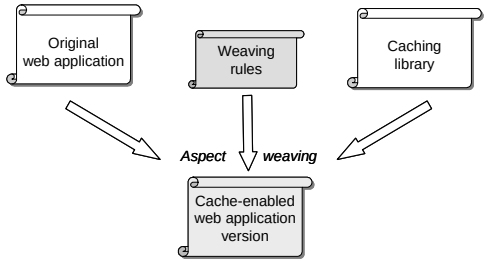


Fig. 2. Aspectizing Caching

3 Dynamic Web Caching

Caching dynamic web content prevents the client from remotely re-accessing the database server to re-execute SQL queries, and from regenerating dynamic web pages on the application server. In this section, we first present the principles involved in designing a dynamic web cache, including our logic to ensure consistency of the cached documents. We then give an overview of our implemented system. Concrete details about the implementation based on aspectizing web caching are provided in the next section.

3.1 Designing a Web Cache

Designing a cache for web documents is rendered complicated by the dynamic nature of web applications, requiring mechanisms to maintain consistency between the data and its cached copy. Specifically, dependency needs to be established between requests that read the data in the back-end (read-only requests) and those that make updates to the back-end (write-requests). We divide the design of such

a caching system into the following mechanisms:

- Cache checks. Upon a client read-only request, the cache is first checked to look up the requested document. In case of a hit, the cached document (e.g. a web page) is simply returned to the client, bypassing the request execution.
- Cache inserts. Upon a miss in the cache during a client read-only request, the request is executed by the application server (and SQL queries are possibly executed on the database server) to dynamically generate a web document that is returned to the client; and a copy of that document is stored in the cache.
- Collecting consistency information. For a read request, we attach the information mapping the underlying database set used in the generation of response to this request (dependency information). Similarly, for a write request, we associate information regarding the database set updated by this request (invalidation information).
- Cache invalidations. Upon a client write request, the cache entries that are affected by the write must be invalidated. This would require making use of the consistency information.

Index: URI (readHandlerName + readHandlerArgs)	Cached web page
URI_1	WebPage1
URI_2	WebPage2
...	...

Index: SQL String	<value vector, URI> pair
$ReadQueryTemplate_1$	<instance $values_{1a}$, URI_1> <instance $values_{1b}$, URI_{41}> <instance $values_{1c}$, URI_{57}>
$ReadQueryTemplate_2$	<instance $values_{2a}$, URI_7>
$ReadQueryTemplate_3$	<instance $values_{3a}$, URI_{12}>
...	...

Fig. 3. Cache Structure

Figure 3 shows the basic structure of our cache. The first table stores the entries of web pages, indexed by URI of the client requests including the request arguments (input info). The second table maintains details about the read-only queries (template + vector of dynamic values = dependency info) used in the formation of the cached pages. When a write query occurs, a query analysis engine determines the set of read queries affected by the update. This information is then used to remove the invalidated entries from the cache.

3.2 Maintaining Cache Consistency

Determining if a client write request invalidates the cached page resulting from a previous client read-only request is equivalent to determining if the set of SQL queries associated with the former request invalidates one of the SQL queries underlying the latter request. For this purpose, our implemented solution includes a query analysis engine that has the task of determining the dependencies between SQL queries. Query analysis has two primary components:

– **Determining possible dependencies between queries**. SQL queries are given as templates (the vector of dynamic values for a particular instance to

be known at run-time). If a read query template shares common tables and columns with an update query template, then a dependency is established.
– **Actual intersection testing to reveal true dependencies**. A true intersection between a read query and an update query (with a dependency established) exists if the update modifies one or more columns in the row(s) being read, and/or results in changing the set of rows satisfying the selection clause of the read query [20].

It is interesting to note that while the first component of this analysis is based on the static portion of the query string (i.e. query template), the second component comes into play at run-time, once we know the actual values used in the selection criteria. For efficiency, our system caches the results of the first component and re-uses them while encountering the same queries again. In practice, there are usually a small fixed number of different query templates, thus, the query analysis cache stabilizes very quickly (Figure 4).

Benchmark	Read queries	Write queries	Number Clients	Time to stabilize
RUBiS	22 types	10 types	1000	< 4 min
TPC-W	10 types	14 types	400	< 1 min

Fig. 4. Query Analysis Cache Statistics for RUBiS and TPC-W

Our analysis engine explores a balance between invalidation precision and its associated evaluation cost, the cost of precision being determined by the detail of query analysis required to extract the relationship needed. The engine supports three cache invalidation policies that increase precision by providing progressively more refined analysis:

1. A simple method is to check if the columns used in the read query are also updated in the write query. This column-only check may result in many *false positive* indications that an intersection exists when, in fact, there is none. E.g., reading then updating column `a` from table `T` creates an intersection, but reading column `a` and updating column `c` does not.
   (a) "`SELECT a FROM T WHERE b=X`" vs "`UPDATE T SET a=new_val...`" *may* intersect if the column updated is `a` (as here) or `b`.
   (b) "`SELECT a FROM T WHERE b=X`" vs "`UPDATE T SET c=new_val...`" does not intersect (assuming `c!= a,b`).
2. To make the test for intersection more precise, selection criteria in the read query's WHERE-clause are matched to values from the write query to see if the same rows are being updated. E.g., if a read's selection clause requires that `T.b=X`, but for the write query `T.b=Y` and $X \neq Y$, then the queries do not intersect.
   (a) "`SELECT a FROM T WHERE b=X`" vs "`UPDATE T SET a=new_val WHERE b=Y`", does not intersect if $X \neq Y$.
3. Invalidates can be made even more precise by executing *extra queries* to retrieve missing data needed to test for intersection. Continuing with the prior example, if the value of the field `T.b` is not specified in the write query itself,

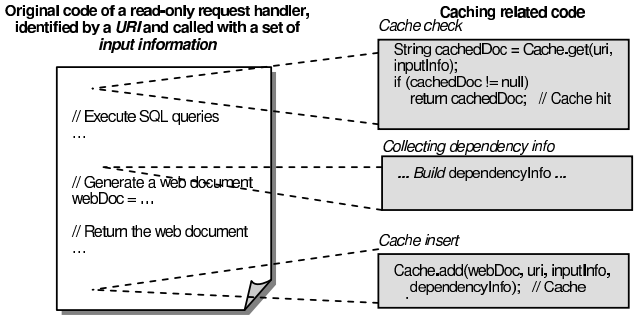


Fig. 5. Caching Read Requests

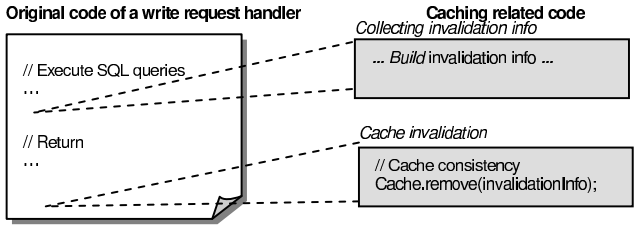


Fig. 6. Handling Write Requests

then an extra query can be made to the database to read the value of `T.b` in the row(s) being updated. This option generates additional queries (by the cache) to the back-end but reduces unnecessary webpage invalidations. E.g.,

(a) "`SELECT a FROM T WHERE b=X`" vs "`UPDATE T SET a=new_val WHERE d=W`", but there is no reference to the value of `b` in the update query.
(b) Therefore, the cache generates a query for column `b` of the row being updated: "`SELECT b FROM T WHERE d=W`".
(c) The read and update queries intersect if the value returned equals `X` (from the read query).

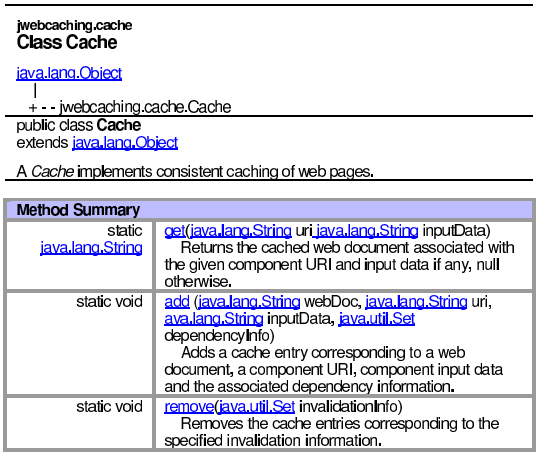
We refer the reader to [20] for detailed descriptions of the engine's handling of various query types for each of the above three cases. The last (and most aggressive) technique which we call the *AC-extraQuery* strategy is used in this study.

Figure 5 and Figure 6 show how collecting dependency and invalidation information, and how cache check, insert and invalidation operations take place within web application request handlers. From the figures, it is clear that to provide consistency, information is gathered both at the front-end (request arguments in the servlet engine) as well as the back-end (queries being shuttled to the database). This is in contrast to caching of SQL query result sets, which requires capturing calls to the database at the JDBC interface only [8].

3.3 Overview of the AutoWebCache system

Our design is called *AutoWebCache*, a system for caching web pages and managing their consistency [3, 20]. In AutoWebCache, the cache is located on (in front of) the

application server (though it could easily be used in a proxy cache formation), and it consists of a set of web pages from read-only requests indexed by the request URI + set of arguments. A page is invalidated if a client update request modifies the data set used to generate the cached page. AutoWebCache uses the most precise cache invalidation strategy discussed prior, namely the *AC-extraQuery* strategy. Web pages resulting from client write requests are not cached.


jwebcaching.cache
Class Cache

java.lang.Object
|
+ - - jwebcaching.cache.Cache

public class **Cache**
extends java.lang.Object

A *Cache* implements consistent caching of web pages.

Method Summary	
static java.lang.String	get(java.lang.String uri java.lang.String inputData) Returns the cached web document associated with the given component URI and input data if any, null otherwise.
static void	add (java.lang.String webDoc, java.lang.String uri, ava.lang.String inputData, java.util.Set dependencyInfo) Adds a cache entry corresponding to a web document, a component URI, component input data and the associated dependency information.
static void	remove(java.util.Set invalidationInfo) Removes the cache entries corresponding to the specified invalidation information.

Fig. 7. Cache API

The main package of the AutoWebCache system is the *jwebcaching.cache* package. It provides several classes, among which the *Cache* class provides the necessary features for cache management, including interaction with the query analysis engine to maintain consistency of the cached web pages. Figure 7 illustrates a part of the API of this class.

4 Aspectizing Web Caching

Aspect-oriented programming (AOP) hands us an efficient tool to perform caching by treating it as a concern that cuts across the application. In this section, we describe our implementation of AutoWebCache, an AOP based caching middleware system. We will also analyze the transparency of the caching aspect with respect to a general application suite.

4.1 Implementing an AOP based caching

AspectJ [2] is an aspect-oriented environment that provides the AOP constructs and set of tools for aspects written in the Java programming language. The AspectJ language exposes a set of join points that are well-defined places in the execution of a Java program flow.

Figure 8 gives an example of a pointcut and advice declaration in the AspectJ language. This example defines a pointcut called *doGetExecution* that designates

```
(a)  aspect ServletExecution {
(b)    // Pointcut definition
(c)    pointcut doGetExecution() :
(d)        execution(
(e)              void HttpServlet+.doGet(
(f)                    HttpServletRequest, HttpServletResponse)) ;
(g)    // Advice definition
(h)    before() : doGetExecution() { ... crosscutting actions  ...}
(i)  }
```

Fig. 8. Pointcut and Advice Examples

the execution of the *doGet* method in the *HttpServlet* class or its subclasses[1] that takes a first argument of type *HttpServletRequest* and a second argument of type *HttpServletResponse* (lines (c)-(f) in Figure 8). This example also defines an advice that executes prior to the specified pointcut (the *doGet* method, line (h) in Figure 8). Please notice that the pointcuts and advices that define the weaving rules to be applied are specified as entities separate from the individual aspect modules. Weaving the final system from individual aspects is performed by the *ajc* tool, the AspectJ compiler.

In order to apply aspect-oriented techniques for caching dynamic web pages in J2EE applications, the following properties are needed:

- The entry and exit points of request handlers in web applications must be well-known points. This is necessary to automatically inject cache check, insert and invalidation operations to those handlers.
- The call to SQL queries that underlie the request handlers in web applications must be well-known points. This is necessary to collect dependency and invalidation information.

4.2 AutoWebCache - an AOP based Web Cache

We implemented AutoWebCache as an AOP-based solution that helps in transparently injecting caching mechanisms to web applications. This involved the following steps:

- **Weaving rules specification** - defines how to integrate the caching aspect into the web application core aspect. The weaving rules specify the points in the application where mechanisms for cache check, insert, invalidation operations etc. need to be injected (see Figure 5 and Figure 6).
- **Aspect weaving** - the process of composing the final cache-enabled system from individual web application and AutoWebCache aspects by following the weaving rules, using the AOP compiler (see Figure 2).

Figure 9 shows how to capture the execution of a Servlet's main method in AspectJ; this is necessary to inject cache checks, inserts and invalidations. Since Java Servlets are defined with a standard API, their main methods are known as being either *doGet* or *doPost* that respectively implement HTTP GET and POST;

[1] The + sign following the *HttpServlet* class name in Figure 8 designates its subclasses.

```
// Pointcut for Servlets' main method
pointcut servletMainMethodExecution(...) :
    execution(
        void HttpServlet+.doGet(
            HttpServletRequest, HttpServletResponse))
    || execution(
        void HttpServlet+.doPost(
            HttpServletRequest, HttpServletResponse));
```

Fig. 9. Capturing Servlets' main method

and the AspectJ's *execution* keyword used in the pointcut captures the execution of those methods [2].

Cache checks and inserts. Figure 10 describes the rules for tackling read-only Servlets. The *around* advice surrounds the normal execution of the main method of a Servlet with cache checks and inserts (the *proceed* keyword calls the normal execution of the method). In case of a cache hit, the normal execution of this Servlet is bypassed. For a cache miss, an entry is added in the cache along with the dependency information associated with this request (c.f., Figure 5).

```
// Advice for read-only requests
around(...) : servletMainMethodExecution (...) {
        // Pre-processing: Cache check
        String cachedDoc;
        cachedDoc = ... call Cache.get of JWebCaching
        if (cachedDoc != null) {
          ... return cachedDoc
        }

        // Normal execution of the request
        proceed(...);

        // Post-processing: Cache insert
        ... call Cache.add of JWebCaching
}
```

Fig. 10. Weaving rules for cache checks and inserts

Cache invalidations. Figure 11 describes an advice that is aimed at tackling write Servlets; it defines the *after* advice that executes following a Servlet's main method. Specifically, it uses the invalidation information attached with this request (c.f., Figure 6) to invalidate the affected cache entries.

```
// Advice for write requests
after(...) : servletMainMethodExecution (...) {
        // Cache invalidation
        ... call Cache.remove of JWebCaching
}
```

Fig. 11. Weaving rules for cache invalidations

[2] In case a Servlet's *doGet* and *doPost* methods are interleaved, it is necessary not to capture the execution of both methods, but only the top-level one. This can be achieved in AspectJ using a *cflowbelow* pointcut (see [17], Chapter 3). For simplicity purposes, we do not use it here.

Collecting consistency information. Figure 12 declares a pointcut that captures calls to read-only and write SQL queries (through standard JDBC API calls, e.g., *executeQuery, executeUpdate*). The *after* advice executes following an SQL query and collects the consistency information - dependency (read query templates + value vectors for a read-only request handler) or invalidation (write query templates + value vectors for a write request), derived from that query.

If a read query is aborted during the formation of response for a client request, the corresponding web page is not stored in the cache. Further, if a write query does not complete successfully, it is not considered for determining the cache entries affected. For simplicity, implementation details concerning these points have been omitted from our presentation.

```
// Pointcut for SQL query calls
pointcut sqlQueryCall() :
    call(ResultSet PreparedStatement.executeQuery())
    || call(int PreparedStatement.executeUpdate());
// Advice for SQL query calls
after() : sqlQueryCall () { ... collect consistency info ...}
```

Fig. 12. Collecting Consistency Information

4.3 Analysing Transparency of AutoWebCache

Caching of dynamic web content can not be considered as an aspect completely orthogonal to the application, in general. In this subsection, we outline some issues that affect the transparency of AutoWebCache with respect to a general application suite.

Capturing Information Flow through various Interfaces. To maintain complete consistency of the cache with the back-end databases, the caching scheme must capture all flow of information in the application, from front-to-back. Such information can flow through various interfaces:

- *Entry and Exit points.* AutoWebCache requires well-defined interfaces for identifying the entry and exit points of a request. In our benchmarks, the Java Servlet APIs provide a standard way to capture entry and exit of a http client request. Further, each cached document is uniquely identified by the URI and Servlet parameters specified in the request.
- *Modification to underlying Data Sets.* When time-lagged weak consistency is employed, once cached, entries are valid until some timeout occurs. To provide a strong consistency of cached documents, however, changes must be tracked on the data used to generate the documents. In our case, we capture modifications to the data sets by capturing the associated SQL requests.
- *Cookies.* Some web applications store part of their request parameters in cookies, instead of specifying them explicitly in the http requests (e.g., the user name and password). In this case, the client includes its cookie [21] in all requests to the

server. A cookie is a small amount of state with no defined structure. Thus, if each web application defines its own ad-hoc cookie structure, transparency is difficult to achieve in AutoWebCache.
- *Multiple Sources of Dynamism.* A dynamic web page can be formed by aggregating data from multiple sources. Currently, AutoWebCache handles dynamism resulting out of SQL queries to a database. However, as long as the interfaces for accessing such sources of dynamism are well-defined, AutoWebCache can be extended easily to provide a high degree of transparency.

The Hidden State Problem. Implied in the design of AutoWebCache is that the http request contains all the information necessary for the servlet to create the web page, thus, identical requests (which will map to the same cache entry) result in the same page being generated. Any other state that affects the web page content is considered *hidden state*. For example, some applications employ randomly generated information for advertisement banners [25]. Another instance is the use of static variables inside the application. In such setups, each subsequent identical http request results in generation of different web pages. Such requests should be marked as uncacheable by the developer.

Use of Application Semantics. For aspect-orientedness to be used, the key semantic concepts must be conveyed via the syntax of the code and, therefore, must be rather straightforward. In some cases, however, understanding the nature of application provides avenues for improving performance of the caching system. For instance, in one of our benchmarks, the TPC-W application, the expensive *Best Seller* web interaction uses a 30 second window allowing dirty reads. In essence, the effects of a change committed to the database by any web interaction which completed less than 30 seconds before the *Best Seller* is permitted to be not reflected in the response page for *Best Seller*. This conforms to clauses 3.1.4.1 and 6.3.3.1 of the TPC-W v1.8 specification [30]. Such concepts form a part of the complex application semantics, and as we demonstrate in the results section, can be quite effective in performance improvement.

5 Evaluation Environment

Test-bed J2EE Web Applications. We tested with the J2EE applications on-line bookstore TPC-W and auction site RUBiS. TPC-W implements an on-line bookstore [30] and defines 14 different interactions among which are accessing a user home page, listing new products and best sellers, registering a new user, updating the shopping cart, ordering. We used an implementation of TPC-W proposed by the University of Wisconsin [18]. RUBiS implements the core functionality of an auction site modeled over eBay [1]. It defines 26 interactions including registering new users, browsing items by category or region, bidding, buying or selling items, and leaving comments. Both TPC-W and RUBiS provide a benchmarking tool that emulates web client behavior and provides statistics (e.g., client response

time). For evaluation, we use the shopping mix for TPCW (80% read requests), and the bidding mix for RUBiS (85% read requests). We vary the client load but the size of the database is fixed.

Client Emulator. Both benchmarks use a client-browser emulator to generate requests. A client session is a sequence of interactions for the same client. For each client session, the client emulator opens a persistent HTTP connection to the Web server and closes it at the end of the session. The average think time between requests (7 sec) and session time (15 min) conform to clauses 5.3.1.1 and 6.2.1.2 of the TPC-W v1.8 specification [30]. All our experiments warm the cache for 15 minutes before collecting statistics over the next 30 minutes.

Software & Hardware. We use the Apache v.1.3.22 web server and the Jakarta Tomcat v3.2.4 servlet engine, with the MySQL v2.04 type 4 JDBC driver, running on Sun JDK 1.4.2. The database is MySQL v.3.23.43-max with MyISAM tables. All machines have an Intel Xeon 2.4GHz CPU, 1GB ECC SDRAM, the 2.4.20 Linux kernel, and a 120GB 7200 rpm disk drive. All machines are connected through a switched 1Gbps Ethernet LAN.

Using this setup, we next analyse the AutoWebCache system, and shed light on some of these important questions:

- What is the effect of AutoWebCache on the performance of an application?
- How does the semantics of an application relate to cache efficiency?
- What is the relative benefit of caching on different read-only requests?
- How much do AOP techniques help in implementing the caching system?

6 Results

In our first experiment, we study the effectiveness of AutoWebCache in reducing the response time of applications. Figure 13 shows the response time for RUBiS, comparing the results of the cache-enabled version (AutoWebCache) with the original application (No cache). Here, RUBiS is running the bidding mix which has

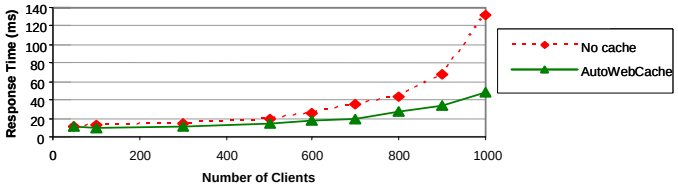


Fig. 13. Response Time for RUBiS - Bidding Mix

updates. Thus, we need to generate cache invalidations to ensure cache consistency. For this mix, the cache hit rate is 54% [3]. We see that AutoWebCache provides a clear performance benefit, improving response time by upto 64%.

[3] All numbers reported here are for the most optimal *AC-extraQuery* cache invalidation strategy of AutoWebCache. See [20] for results comparing different strategies.

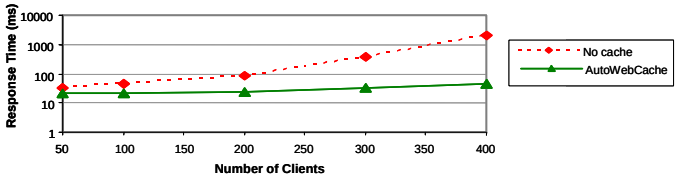


Fig. 14. Response Time for TPC-W - Shopping Mix

Figure 14 shows the results for TPC-W, using the primary reporting mix of shopping which has updates. Please note the log scale of the y-axis. From the graph, we again see that AutoWebCache version of the application has significantly faster response times than the No cache version. In this case, the response time is reduced by up to 98%, and the cache hit rate is 43%. The overhead of processing cache lookups can be measured by forcing a cache miss on every lookup. The performance difference to NoCache is negligible (not distinguishable at the millisecond scale) so it is not shown in the graph.

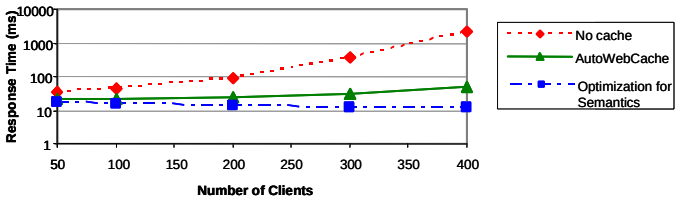


Fig. 15. Cache Improvement in TPC-W based on Application Semantics

In our second experiment, we present how knowledge of the application semantics can help in improving the efficiency of AutoWebCache. In TPC-W application, the expensive `BestSeller` request uses a 30 second window allowing dirty reads, permitting those changes committed to the database less than 30 seconds before this request to be not reflected in the response (c.f., Section 4.3). Making use of this semantics, the best seller pages were marked cacheable for a full 30 second window. The performance improvement with this optimization is shown in Figure 15.

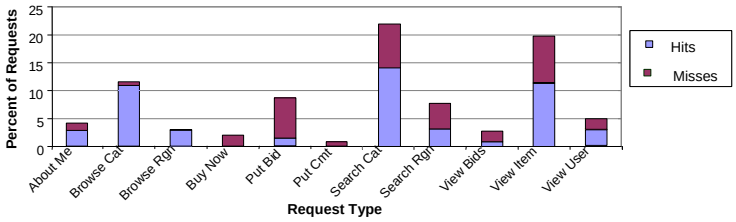


Fig. 16. Relative Benefits for different Requests in RUBiS

In our next experiment, we analyze the relative benefit of caching on different individual read-only requests. Figure 16 shows that for RUBis (with 1000 clients),

as expected, requests benefit by varying degree using the AutoWebCache system. Requests `BrowseCategories` and `BrowseRegions` have an almost 100% hit rate, while requests `BuyNow` and `PutComment` have the least cache hit ratios. While most of the misses in the last two categories were cold misses,[4] for `ViewItem` and `ViewBids`, most of the misses were due to invalidation of the cached entries.

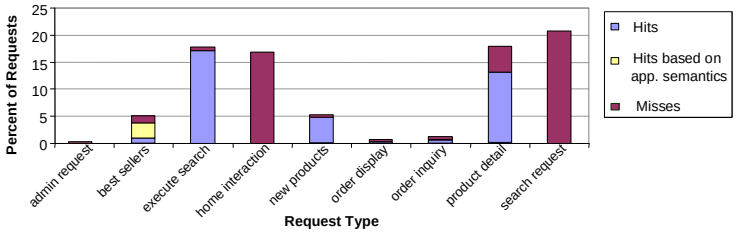


Fig. 17. Relative Benefits for different Requests in TPCW

Figure 17 shows the relative benefits experienced by different requests for TPCW, running with 400 clients. However, there are two differences in this graph from the one we obtained for RUBiS. Firstly, in the case of TPCW, two requests (unlike any in RUBiS), `SearchRequest` and `HomeInteraction` were explicity marked uncacheable because they use a random number generator to produce advertisement banners. Secondly, most of the hits for `BestSeller` request were obtained using a 30 second window for invalidation (described earlier). Such application semantics were not used for any request in RUBiS.

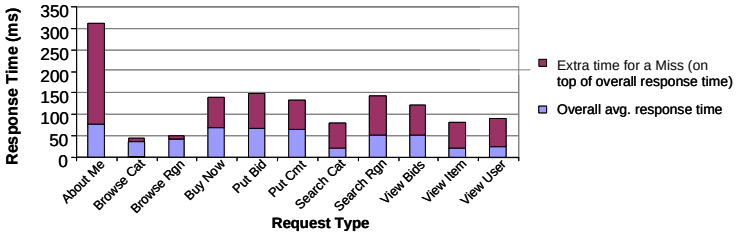


Fig. 18. Breakdown of different Requests in RUBiS w.r.t. Response Time

Figures 18 and 19 report the improvement in response times of individual requests with AutoWebCache, for RUBiS and TPCW with 1000 and 400 clients, respectively. For each request, the graphs show the average extra time required to generate the response for that request in case of a cache miss. Hence, for a miss, the response time for a request is the sum of the two components. In the case of RUBiS, `AboutMe` has high penalty for a miss. However, this is compensated by a high hit rate for this request. Same arguments can be applied for `BestSeller`, `ExecuteSearch` and `NewProducts` requests in TPCW. Also, since the requests `SearchRequest` and `HomeInteraction` have low response times, marking them uncacheable does not impact the performance of AutoWebCache a great deal.

[4] Hits for these requests require the same customer *and* item as a previous request.

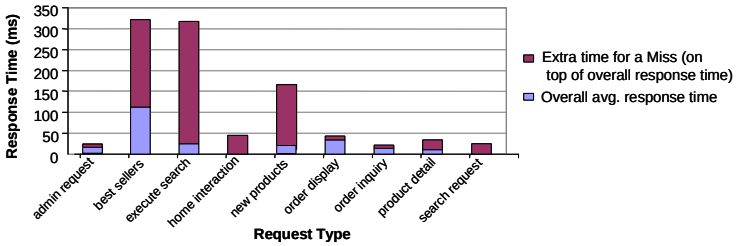


Fig. 19. Breakdown of different Requests in TPCW w.r.t. Response Time

	Web application		Caching library		AOP-based caching	
Application	# Java classes	Java code size	# Java classes	Java code size	# AspectJ files (weaving rules)	Size of AspectJ code
TPC-W	46	12K lines	13	4.6K lines	1	150 lines
RUBiS	25	5.8K lines				

Fig. 20. Web App & Cache Library Code Size vs. Aspect-J Code Size

Figure 20 compares the code size of the individual aspects, the TPC-W and RUBiS testbed applications and the JWebCaching library. Most of the code for the AutoWebCache system, including the query analysis engine, lies in the JWebCaching library. This library implements the cache interfaces and can be reused for various applications. Size of code written in AspectJ for weaving caching into the application is much smaller. Thus, it is easy to maintain and customize for different applications.

7 Discussion

A goal in web cache research has been to develop designs that are completely transparent to the application yet supports strong consistency. Complete transparency means that no effort is required from the application programmer to achieve caching - such a cache would be easy to add. Support for strong consistency means the cache can ensure it is always synchronized with the state of the persistent backing store - such a cache would have a wide audience. Caching of static content achieves both goals. Strong consistency is trivial by the fact that the content does not change. Transparency is easily achieved as the final content can be captured at a well-known point - while being sent as the response to a client's request.

The complexity of maintaining consistency in our case is due to caching *derived data*. We call data such as web pages derived because they are obtained using some set of data in the persistent store of the application (e.g., rows in the database tables). In contrast, non-derived data objects map directly to *unique* items in the persistent store. Thus, checking for inclusion in the cache is a simple matter of checking for the existence of a unique global identifier (e.g., a simple id). When caching derived data, however, the mapping relationship is obscured. Complexities arise as more than one document can depend on the same field in the database.

Also, dynamic web pages can aggregate data from multiple databases. Therefore, detecting if a change to a database affects a web page involves testing for inclusion of the changes in each page's input set.

Aspect-oriented programming is an efficient tool to capture the information flow in an application and can be used to inject the caching calls at appropriate points. Working with AOP gives several benefits:

Modularity. Separation of concerns is inherent to AOP-based systems. The implementation of each individual aspect (e.g., the J2EE web application and the logic for caching dynamic content) may evolve separately without inducing a change in the implementation of the others.

Generality. The AutoWebCache prototype uses AOP to add caching of dynamic web pages to a Servlet-based web application that interfaces a database with JDBC. This methodology is general enough to encompass other sources of dynamic data. Specifically, individual aspects can be developed separately for each source and then woven together.

Transparency. Any modification/extension to the application interfaces is captured by making appropriate changes in the pointcut specifications, and not the way individual aspects have been developed or woven. This provides a clean way to make caching look oblivious to the developer.

Our AOP-based framework combines simplicity with flexibility to achieve a good level of transparency. Let us compare our technique with a compiler-based approach as in [8]. The compiler does a similar query analysis at compile time and embeds the results for simple look-up at run-time. The proposed AutoWebCache system also achieves almost zero run-time analysis overhead via result caching [3], but is much easier to develop than compiler techniques, making use of AOP tools. Another subtle advantage of our approach is that it is robust even if the SQL queries are dynamically formed, as it captures the run-time value of the string at the point of SQL call. For a compiler, query strings must be statically available. This assumption might not hold for real-life, complex applications.

We believe that achieving the simultaneous goals of complete transparency and strong consistency in web caching is not possible for the general case. The key problem is in automatically verifying that no essential data in an application needed for caching flows through unexpected interfaces and, thus, elude the consistency logic. Cookies, randomly generated data and application semantics are some examples of this phenomena from our benchmarks. If an application presents a fairly orthogonal caching aspect, AutoWebCache would require only minimal developer intervention. If not, a special weaving rule would be constructed for each non-orthogonal concept. In the worst case, AOP would extend only modularity as a benefit, same as that offered by object-oriented techniques.

8 Related Work

Caching of dynamic content with weak consistency can achieve transparency because, as for static content, no information is needed to synchronize the cached documents with the backing store. Typically, pages can be set to timeout so that

the cache content is periodically refreshed. CachePortal [4] has a unique form of weak, time-lagged consistency. It relies on timestamps and HTTP logs to conservatively determine which pages to invalidate. Inconsistencies can exist for a time between the cache and the backing store.

While caching the contents of the persistent store (non-derived data) directly, a high degree of transparency with strong consistency can be achieved. Examples include caching direct copies of raw DBMS tables [29] or caching copies of persistent Java objects [13]. A framework where business rule SQL query result sets are cached is presented in [8, 12]. As with our work, strong consistency is maintained through complex analysis of the SQL queries. A high degree of transparency is achieved through the compiler-based solution to insert the cache API calls tuned to the Websphere environment. In contrast, our work uses much simpler AOP tools.

Examples of caching derived data with strong consistency suffer from a low level of transparency that requires considerable developer input about request structure or dependencies. DynamicWeb [10] provides a strongly consistent web page cache, but not transparently as developers must define the dependencies between events, e.g., read and write queries. Similarly, form-based proxy caching [19] of web pages requires developers to pre-define configurations of web page formats. Weave [33] requires the programmer to use a specialized language to describe dynamic web pages and event handlers to specify invalidations. Various commercial solutions such as SpiderCache [26], Xcache [32], and Oracle9iAS [22] provide an event API to the developers to add consistency management.

The current prototype of AutoWebCache is implemented as a generic solution for a J2EE web application that uses Servlets embedding SQL queries based on JDBC [27] since this pattern is widely used in many J2EE applications [5]. It can be easily extended to include other sources of dynamism, as well as other ways of forming dynamic content, such as PHP [31]. Furthermore, the proposed caching solution is completely transparent when all database updates go through the server-side application. However, if some updates are directly performed on the database, transparency is difficult to achieve. A possible solution is to extend the caching system with an API similar to the ones provided by the DynamicWeb and Weave systems to allow an external entity to invalidate cache entries [10, 33]. This external entity could, for instance, work through database triggers.

AOP techniques were experimented for profiling [7], persistence [23], distribution [14], web cache pre-fetching [24], caching static content [17], caching (non-derived) Java objects [13], and also for transactions [16] where the authors conclude that, as for consistent caching of dynamic web content, transactions can not be aspectized in general.

9 Conclusions

In this paper, we presented AutoWebCache, a middleware system for adding caching of dynamic content transparently to J2EE server-side applications having a backend database. Caching fully-formed webpages reduces the work at both

the increasingly costly business logic tier as well as the back-end database tier. We first outlined the principles involved in caching dynamic web content, including the logic to ensure consistency of the cached documents. Thereafter, we demonstrated the use of aspect-oriented techniques to implement our system. We showed how aspect-oriented techniques improve modularity and transparency of the entire solution.

Using two standard J2EE web benchmarks, RUBiS and TPC-W, we evaluated AutoWebCache along various dimensions. First, we studied the effectiveness of AutoWebCache in reducing the response time of applications. Second, we analyzed the transparency of our system for a general application suite. We argued that for the general case, issues may arise when caching at the front-end as dynamic web pages can aggregate data from multiple sources and also some sources might not have sufficiently structured interfaces for exchanging the information necessary for tracking coherency. Furthermore, we showed that knowledge about application semantics can improve the efficiency of caching.

Our work presents itself several avenues for extension. A database query-results cache is complementary to webpage caching. Complex SQL queries that cannot be efficiently parsed for coherency dependency information (e.g., range queries) can be declared uncacheable at the front-end webpage cache but have its result sets cached at the back-end, thus, reducing the database costs if not the business logic costs for those requests. We also want to extend the AutoWebCache system to incorporate sources of dynamism other than SQL queries, and study their transparency w.r.t. AOP. Finally, we want to analyze the effect of varying cache size on the hit rates of requests and investigate different cache replacement strategies in this context.

References

1. C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani and W. Zwaenepoel: Specification and Implementation of Dynamic Web Site Benchmarks. IEEE 5th Annual Workshop on Workload Characterization (WWC-5), Austin, TX, USA, Nov. 2002. http://rubis.objectweb.org
2. AspectJ 1.1, 2004. http://www.eclipse.org/aspectj/
3. S. Bouchenak, A. Cox, S. Dropsho, S. Mittal and W. Zwaenepoel: Caching Dynamic Web Content in J2EE Applications. EPFL Tech. Report IC/2004/82, Oct. 2004.
4. K. S. Candan, W.S. Li, Q. Luo, W.P. Hsiung and D. Agrawal: Enabling Dynamic Content Caching for Database-driven Web Sites. ACM SIGMOD'2001, Santa Barbara, CA, USA, 2001.
5. R. Cattell and J. Inscore: J2EE Technology in Practice: Building Business Applications with the Java 2 Platform, Enterprise Edition. Pearson Education, 2001.
6. A. Colyer: Implementing Caching with AOP. TheServerSide.COM, June 2004. http://www.theserverside.com/blogs/showblog.tss?id=AspectJCaching
7. J. Davies, N. Huismans, R. Slaney, S. Whiting, M. Webster and R. Berry: Aspect Oriented Profiler. 2nd Intl. Conference on AOSD, Boston, USA, Mar. 2003.
8. L. Denagro, A. Iyengar, I. Lipkind and I. Rouvellou: A Middleware System Which Intelligently Caches Query Results. Middleware Conference, NY, USA, Apr. 2000.
9. A. Feldmann, R. Cceres, F. Douglis, G. Glass and M. Rabinovich: Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments. IEEE Conference on Computer Communications (INFOCOM), New York, Mar. 1999.

10. A. Iyengar and J. Challenger: Improving Web Server Performance by Caching Dynamic Data. In proceedings of USITS'97, Monterey, CA, USA, Dec. 1997.
11. A. Iyengar, E. MarcNair and T. Nguyen: An Analysis of Web Server Performance. IEEE Global Telecommunications Conference (GLOBECOM), Phoenix, 1997.
12. A. Iyengar and J. Challenger: Data Update Propagation: A Method for Determining How Changes to Underlying Data Affect Cached Objects on the Web. IBM Technical Report RC 21093(94368), IBM Research Division, Feb. 1998.
13. JBoss Inc. JBossCache. http://www.jboss.org/products/jbosscache
14. M. A. Kersten and G. C. Murphy: Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming. ACM Conference on OOPSLA, Denver, Colorado, USA; Nov. 1999.
15. G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier and J. Irwin: Aspect-Oriented Programming. ECOOP'97, Jyvskyl, Finland.
16. J. Kienzle and R. Guerraoui: AOP: Does it Make Sense? The Case of Concurrency and Failures. In proceedings of ECOOP'2002, Mlaga, Spain, June 2002.
17. R. Laddad: AspectJ in Action - Practical Aspect-Oriented Programming. Manning Publications, 2003.
18. M. H. Lipasti: Java TPC-W Implementation Distribution. htt://www.ece.wisc.edu/ pharm/tpcw.shtml
19. Q. Luo, J. F. Naughton: Form-Based Proxy Caching for Database-Backend Web Sites. 27th Very Large Data Bases Conference (VLDB'2001), Roma, Italy, 2001.
20. S. Mittal: A Consistent and Transparent Solution for Caching Dynamic Web Content. Masters thesis, Rice University, 2004. http://www.cs.rice.edu/ mittal/presentations/thesis_mittal.pdf
21. Netscape. Persistent Client State - HTTP Cookies. http://wp.netscape.com/newsref/std/cookie_spec.html
22. Oracle. Oracle9iAS Caching Solutions. Oracle Technical White Paper, Dec. 2001. http://otn.oracle.com/products/ias/web_cache/pdf/9ias_caching_twp.pdf
23. A. Rashid and R. Chitchyan: Persistence as an Aspect. 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, 2003.
24. M. Sgura-Devillechaise, J. M. Menaud, G. Muller and J. Lawall: Web Cache Prefetching as an Aspect : Towards a Dynamic-Weaving Based Solution. 2nd International Conference on AOSD, Boston, USA, Mar. 2003.
25. S. Sol and G. Berznieks: Instant Web Scripts with Cgi Perl. M & T Books, 1996.
26. Spider Software. SpiderCache Enterprise 2.0:Dynamic Content Delivered Faster. Spider Software Technical White Paper, Sep. 2001. http://www.spidercache.com/
27. Sun Microsystems. Enterprise Applications with J2EE Platform, 2nd Edition. http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/
28. Sun Microsystems. Java DataBase Connection (JDBC). http://java.sun.com/jdbc/
29. TimesTen. TimesTen Real-Time Event Processing System. TimesTen White Paper, 2003. http://www.timesten.com
30. Transation Processing Performance Council. TPC-W: a transactional web e-Commerce benchmark. http://www.tpc.org/tpcw/
31. PHP: Hypertext Preprocessor. Visual PHP Web Development and Web Reporting. http://www.yessoftware.com/content_simple.php?content_id=php_org.
32. XCache Technologies. XCache Overview. 2004. http://www.xcache.com
33. K. Yagoub, D. Florescu, V. Issarny and P. Valduriez: Caching Strategies for Data-Intensive Web Sites. 26th Very Large Databases Conference (VLDB), Egypt, 2000.